Using Rules to Generate and Execute Workflows in Smart Factories

Dörthe Arndt, Joachim Van Herwegen, Ruben Verborgh,
Erik Mannens, and Rik Van de Walle

Ghent University - iMinds - Data Science Lab
Sint-Pietersnieuwstraat 41, B-9000 Ghent, Belgium
doerthe.arndt@ugent.be

Abstract. In modern factories, different machines and devices offering their services, such as producing parts or simply providing information, become more and more important. The number and diversity of such devices is increasing and the task of combining available resources into workflows becomes a challenge which can hardly be handled by a human user. In this paper we describe how we use RESTdesc, a formalism to semantically describe possible actions of RESTful Web APIs via existential rules to automatically generate and execute such workflows. Our approach makes use of Notation3 reasoners and their ability to produce proofs. These proofs are interpreted as workflow descriptions which can be easily executed and updated. The latter makes our approach very adaptable to unforeseen situations. By using one rule per possible API call, our system is very modular and easy to maintain; services can be readily added or removed. Our implementation shows how the use of rule-based reasoning can significantly improve the daily work in today's factories.

Keywords: Notation3, RESTdesc, Existential Rules, Industry 4.0

1 Business Case

Our business case applies to workflows in factories. Modern factories rely on a huge variety of services provided by different computers; there are sensors which measure the temperature of a machine, there are interfaces to start and stop machines, there are different sources to share knowledge, computers to communicate with human users, and many more. Access to all these services is given through RESTful Web APIs whose functioning is semantically described. A workflow system should take these descriptions into account, properly combine them, and execute the services when it comes to the task of fulfilling a desired goal. It should be flexible enough to change workflows when a service does not behave as expected or simply fails, and it should support all kinds of reachable goals which could be semantically expressed within the language. Such an adaptable goal-driven workflow system facilitates factory work and organization:

– Existing software services can be used in an optimal way; if a new task needs to be solved, the system is aware of existing services and can use them. No service gets forgotten. Unnecessary expenses for the implementation of new use case specific components or machines can be avoided.

- The maintenance of the workflow system is easy. If a new component is bought, it can easily be added to the workflow system by adding its functional description. If a component will not be used anymore, its description can be simply removed. No time is lost by editing rather complex detailed workflows.
- If a component temporarily does not work, another service (or the combination of several services) can cover for that. So the daily work in the factory does not get disturbed by the failure of a single component.
- Users have the possibility to define their own instant goals. Even workflows which do not get repeated very often (like for example accessing different sources to get data about the state of a specific single component of a machine) are supported.
- Knowledge can be shared and accessed in a more efficient way. Data in factories is often stored in different computer systems or databases. An adaptive system as described above is aware of the sources and can compose and execute workflows to access those as needed. No time is lost by searching for information.

2 Technological Challenges

A goal driven adaptive workflow system as described above should:

Semantics use the knowledge at its disposal (description of services, background knowledge about infrastructure, machines and workers) to compose and execute a plan in order to reach a given goal;

Adaptivity adapt the plan if (for example due to service failure) any service does not behave as expected;

Configuration allow to add or remove service descriptions from the system at configuration time;

Flexibility immediately support all viable goals the user could express in its underlying language;

Scalability be able to deal with a big amount of possible services and knowledge.

There are several options to implement workflow systems. In industrial contexts, process-oriented approaches are very common. Systems like for example jBPM [3] or Activiti [1] enable the user to express whole processes using Business Process Model and Notation (BPMN) [2]. While such systems are a good solution for fixed and error-resitent workflows which get repeated frequently, they are not suited for the use case described above: they are not able to autonomously compose workflows, which insted have to be specified by the user. They do not take context information like working schedules in factories or functioning of machines into account (*Semantics*). If something in a workflow goes wrong, e.g., if a component fails, such a system can only react on that if that case has been considered in the workflow description (*Adaptivity*). If services get added or removed from the system, all workflows making use of this particular component need to be edited, because the system is not able to adapt to that case (*Configuration*). If a new goal becomes relevant to the system, the workflow towards that particular goal needs to be explicitly modeled, even if all relevant components are already used in other contexts (*Flexibility*). As the workflows are known beforehand, these kinds of systems can be optimised; therefore most of them scale very well (*Scalability*).

```
@prefix : <http://facts4workers.eu/>.
@prefix http: <http://www.w3.org/2011/http#>.
@prefix tmpl: <http://purl.org/restdesc/http-template#>.

{     ?part  :partId ?id. }
=>
{   _:request http:methodName "GET";
              tmpl:requestURI ("http://f4w.eu/parameters/" ?id);
              http:resp [ http:body _:parameters ].
      ?part  :productionParameters _:parameters. }.
```

Listing 1. RESTdesc description for an API call to retrieve the production parameters for a part.

Given the requirements above, especially the need for flexibility and configurability, we opted for a rule-based solution. In the proposed implementation, each possible operation of a service is described by a rule taking into account the data and resources needed to apply a service, a description on how it can be applied, and the situation which results from its execution. We then make use of the well-studied mechanisms employed in rule-based reasoning. Rules and facts get combined into a formal proof for a desired goal. This can then serve as a plan which can be updated after every service-call execution. Our approach is thereby highly adaptive, as will be pointed out in the following sections.

3 Rule-based Solution

In this section our rule-based solution is detailed. First, we introduce RESTdesc [9], the description format for RESTful Web APIs employed in our solution. After that we explain how RESTdesc descriptions can be combined into proofs which then serve as plans towards a goal. These plans can be executed and updated. In the third part of this section we illustrate this in more detail and explain how our framework adapts in cases where services do not behave as expected. Our approach is based on an earlier paper [7], where a more formal consideration of the algorithm described below can be found.

3.1 Describing RESTful Web APIs

RESTdesc is a way to formally describe the functionality of RESTful Web APIs in terms of rules. These are expressed in Notation3 (N3) [5], a logic which forms a superset of RDF and extends the RDF data model by functional predicates, logical operators (in particular implication), and the option to refer formulas (graphs). N3 distinguishes between two kinds of variables, the universal variable starting with '`?`' and the existential variable or blank node beginning with '`_:`'. These variables are implicitly existentially and universally, resp., quantified [4]. N3 supports blank nodes in the conclusion of a rule, and allows thereby reasoning with existential rules.

The latter is one of the reasons why we use N3 for our purposes; a RESTdesc description is an existential rule describing a possible API operation. Each such rule has the raw structure

```
{ precondition. } => { HTTP-request. postcondition.}
```

where the three parts *precondition*, *HTTP-request* and *postcondition* are as follows:

1. The *precondition* specifies the resources needed to perform the operation described. Such a description may not include existential variables.
2. The *HTTP-request description* explains by which exact request the desired result can be obtained. The request itself is represented by a new existential variable, all its other variables should also be contained in the precondition.
3. The *postcondition* describes the results that the execution of the operation provides. All universal variables it contains should also be present in the precondition. The postcondition may additionally contain existential variables.

Note that the conditions ensure that all universal variables occurring in the consequence of a rule are also present in the antecedent; RESTdesc descriptions do not introduce new universally quantified variables. By choosing an existential variable to represent the request itself, we express that, given the circumstances described in the pre-condition, the existence of such a call can be guaranteed. The existential variables which are only allowed in the postcondition fulfill a special purpose: they express the knowledge of whose existence we know *before* executing the call, but whose exact form we can only know *after* that. These variables get instantiated by performing the call.

To better understand this idea we take a closer look at the formula displayed in Listing 1. The formula describes an API operation which, given a certain part, can be performed to retrieve its production parameters like for example its measurements and the material it is made of. According to the *precondition* (line 5) we need a part and its part-id to be able to execute the described API-operation. With this information an HTTP-GET-request to the URI `http://f4w.eu/parameters/?id` can be constructed with `?id` being a placeholder for the actual part-id[1] (*HTTP-request description*, line 7–8). In lines 9–10, the consequences of the execution of such an API call are expressed (*postcondition*). As a response, the call gets the production parameters of the part with this particular part id. In the description, these parameters are represented by a blank node, `_:parameters`, an existential variable which is quantified within the consequence of the rule [4]. Given a part and its id, we do know of the existence of these parameters and that they can be retrieved by executing the call, their exact nature and content is only known after the execution of the call.

3.2 A proof as a workflow description

Having introduced the basic idea of RESTdesc descriptions in the last section, we now explain how these can be combined into a workflow. Here we make use of the fact that the descriptions are simple rules and can as such be applied and combined by a rule-based reasoner to verify a desired goal. N3 reasoners work in a goal-driven manner; provided with facts, rules, and a goal, they will try to find evidence for the latter and, in case of success, provide a formal proof. As in such a proof all rules which were used are also displayed, we can interpret it as a workflow description and follow it rule-by-rule.

We clarify this by an example. Additionally to Listing 1, we take another RESTdesc description into account; Listing 2 displays a formula which explains how, given the

[1] Here the formula makes use of an HTTP-template, see *http://purl.org/restdesc/http-template*.

```
1  @prefix : <http://facts4workers.eu/>.
2  @prefix http: <http://www.w3.org/2011/http#>.
3
4  { ?part :productionParameters ?parameters. }
5  =>
6  { _:request http:methodName "POST";
7              http:requestURI "http://f4w.eu/machine1/produce";
8              http:body ?parameters;
9              http:resp [ http:body _:newPartInstance ].
10   _:newPartInstance :instanceOf ?part.      }.
```

Listing 2. RESTdesc description for an API call which starts the production of a part.

production parameters of a certain part, a machine can be called to produce that part. We furthermore have a knowledge base which contains, among other information, the fact that a `cog` is a type of part which can be produced and that the internal id of `cog` is 27:

```
:cog :partId 27.                                                    (1)
```

We now want to produce an instance of such a cog and invoke the reasoner with a goal:

```
{ ?x instanceOf :cog. }=>{ ?x :instanceOf :cog. }.                  (2)
```

In N3 goals are given to the reasoner as simple rules. These are used as filters, i.e. in a concrete proof the application of these rules will appear as the very last step. As we do a kind of query here, premise and conclusion of the rule do not differ.

Taking a closer look at the formulas given, a possible workflow towards our goal could be: we use the knowledge from Formula 1 which provides us with all information needed to apply the API call from Listing 1. That call returns the production parameters for a `:cog`. These parameters can be used in a second step; the execution of the API call in Listing 2, which will then initialize the production of an instance of our part `:cog`.

This plan is also displayed in the (shortened) proof of Listing 3, which was produced by the EYE reasoner [8]. The proof consists of different *lemmas*, each being a reasoning step. Lemmas 4–7 [*lines 32–35*] contain the steps of extraction, parsing and conjunction elimination applied to the input formulas. The filenames used for the source files indicate to which formula each lemma refers. Lemma 3 [*line 23*] is an inference applying the RESTdesc description from Listing 1 to the knowledge base triple from Formula 1. The object of the predicate `r:gives` displays the instantiated consequence of that inference; we see a description of the API call which returns the production parameters of `:cog`. As the HTTP-request is free of variables or *sufficiently specified* [7], we could make a GET-request to the URI *http://f4w.eu/parameters/27* which would, if the API works as described, provide us with the production parameters, here expressed by the (renamed) blank node `_:sk_2`. The inference step in Lemma 2 [*line 14*] now makes use of the knowledge derived in Lemma 3 and applies the RESTdesc rule from Listing 2 to it. Here the object of the `r:gives` is formed by the instantiated description of the HTTP-request which starts the actual production of an instance of our part `:cog`. Note that this call uses of the production parameters `_:sk_2` whose exact appearance is not known at this point, so this request cannot be executed. In Lemma 1 [*line 10*] the goal rule (Formula 2) is applied to the consequences of Lemma 2 and leads to the result of the proof [*line 6*].

```
@prefix : <http://facts4workers.eu/>.
@prefix http: <http://www.w3.org/2011/http#>.
@prefix tmpl: <http://purl.org/restdesc/http-template#>.
@prefix r: <http://www.w3.org/2000/10/swap/reason#>.

[] a r:Proof, r:Conjunction;
           r:component <#lemma1>;
           r:gives { _:sk_5 :instanceOf :cog. }.

<#lemma1> a r:Inference;
           r:gives {_:sk_5 :instanceOf :cog};
           r:evidence (<#lemma2>); r:rule <#lemma4>.

<#lemma2> a r:Inference;
           r:gives {_:sk_3 http:methodName "POST".
                    _:sk_3 http:requestURI "http://f4w.eu/produce".
                    _:sk_3 http:body _:sk_2.
                    _:sk_3 http:resp _:sk_4.
                    _:sk_4 http:body _:sk_5.
                    _:sk_5 :instanceOf :cog};
           r:evidence (<#lemma3>); r:rule <#lemma5>.

<#lemma3> a r:Inference;
           r:gives {_:sk_0 http:methodName "GET".
                    _:sk_0 tmpl:requestURI
                                ("http://f4w.eu/parameters/" "27").
                    _:sk_0 http:resp _:sk_1.
                    _:sk_1 http:body _:sk_2.
                     :cog :productionParameters _:sk_2};
           r:evidence (<#lemma6>); r:rule <#lemma7>.

<#lemma4> a r:Extraction; r:because [a r:Parsing; r:source <goal.n3>].
<#lemma5> a r:Extraction; r:because [a r:Parsing; r:source <api2.n3>].
<#lemma6> a r:Extraction; r:because [a r:Parsing; r:source <know.n3>].
<#lemma7> a r:Extraction; r:because [a r:Parsing; r:source <api1.n3>].
```

Listing 3. Example API composition proof.

The proof displays the dependencies between lemmas—and thereby also between API calls—and it contains concrete descriptions of the HTTP-requests which have to be executed to achieve the goal. Thus, this proof can be understood as a workflow description. Given the above, we would first execute the instantiated call from Lemma 3, the next step would be Lemma 2 leading then to our goal, the production of a `:cog`.

3.3 Dealing with alternatives

After having seen in the last section how a formal proof serves as a workflow description, we explain here how we *use* these descriptions. The basic idea of our algorithm is to repeat the following steps: first we find the HTTP-requests which are *sufficiently instantiated*, as a second step we execute one of them, and then add a representation of the execution result in N3 to our knowledge base. After that we generate a new plan. This new plan can either be shorter than the previous one, then we repeat the procedure, or it could be that the new information did not change the proof, in that case the last

service used did not behave as expected, we therefore remove it from our knowledge base and try to find a new proof to then again follow these steps. The algorithm ends if either no proof can be generated in the first step (*failure*) or if the generated proof does not contain HTTP-request descriptions (*success*).

We clarify this procedure following our previous example: the proof in Listing 3 contains a *sufficiently specified* HTTP-request, the one in Lemma 3. We execute the GET-request to *http://f4w.eu/parameters/27*. If the call behaves as described we get parameter information back, for example radius, width and material:

```
:cog :productionParameters
        {:parameters :radius "2cm"; :width "0.5cm"; :material :steel.}.
```

After adding that formula to the knowledge base and starting the reasoner again, we get a new proof but this time with one inference step less as the application of the rule in Listing 1 to retrieve production parameters is no longer needed. The new proof makes use of the triple above. In the shorter version the value for the body `_:sk_2` in Lemma 2 will be replaced by the actual parameters, `{:parameters :radius "2cm"; :width "0.5cm"; :material :steel.}`. This then makes the description in Lemma 2 *sufficiently specified*. In a next step we could execute the POST-request. If this call also ends as expected the goal is reached.

The execution of the request in Lemma 3 could also lead to a different result: it could for example already start the machine and produce the part. In that case the proof which is generated after adding a representation of that event to the knowledge base does not include API descriptions any more, the process would also successfully end. A third possibility is that the call returns data which is useless for our purpose or no data at all, for example, because the service is temporarily out of order. In that case we simply remove the description of Listing 1 from the current knowledge and then try to find another plan towards the goal which does not include the service which failed. If there exist other APIs which return the production parameters of a cog or if there is another API to cause its production which relies on different data which can also be retrieved via a described API call, this will be found by the reasoner. It will then display a new plan which can be executed.

4 Status

The example presented in this paper consists of simplified formulas employed in the use cases we currently tackle in our project FACTS4WORKERS[2] (F4W). This Horizon 2020 project is part of the Factories of the Future PPP[3] and focuses on the needs of factory workers and the question how their daily work can be improved by means of technology. One important aspect here is to use and combine the technology which is already present in the factory in an optimal way and our implementation is aiming to solve exactly this problem. We were able to apply our framework on several of the identified challenges[4] and are working on the others at this moment.

[2] *http://facts4workers.eu/*

[3] *http://ec.europa.eu/research/industrial_technologies/factories-of-the-future_en.html*

[4] *http://facts4workers.eu/wp-content/uploads/2015/12/F4W_D1.2_Requirements.pdf*.

The implementations already done lead to promising results: we are able to incorporate additional semantic information and to apply this in the reasoning process which generates the workflows. The workflow composition itself benefits from the developments achieved by rule-based reasoning. As the RESTful API operations are described by simple rules, which are written independently from each other, it is easy to add or remove services from the knowledge base. As the workflow is recomposed after every service execution, our implementation can easily adapt to unforeseen situations and update or change its plan. The fact that the workflows are not fixed furthermore guarantees that new viable goals are supported immediately. The main challenge of our approach remains scalability: first tests presented in our previous paper [7] show that the reasoner we use—EYE [8]—scales very well for short paths even with bigger amounts of RESTdesc rules. The reasoner additionally supports different reasoning strategies, e.g., *linear logic*, where every rule can only be applied once, or *existing-path*, a strategy which extends EYE's basic Euler path detection such that it also tests for homomorphisms of existential graph patterns (see also [6]). Both strategies improve the performance of reasoning with existential rules. Their optimal usage will be part of future research.

All tests done so far on our framework were executed on theoretical set-ups and not in a real-world environment; therefore another important next step after finishing the implementation of all the use cases is to test our framework together with our industrial partners in the factory set-ups where it is meant to operate. By doing this we will perform one further step from our theoretical framework towards a real-life application which will improve the daily work in a smart factory.

References

1. Activiti BPM Platform. *http://activiti.org/*, accessed: 2016-05-28
2. Business Process Model and Notation (BPMN). *http://www.omg.org/spec/BPMN/2.0/PDF/*, accessed: 2016-05-28
3. RedHat JBoss jBPM. *http://www.jbpm.org/*, accessed: 2016-05-28
4. Arndt, D., Verborgh, R., De Roo, J., Sun, H., Mannens, E., Van de Walle, R.: Semantics of Notation3 logic: A solution for implicit quantification. In: Bassiliades, N., Gottlob, G., Sadri, F., Paschke, A., Roman, D. (eds.) Rule Technologies: Foundations, Tools, and Applications. Lecture Notes in Computer Science, vol. 9202, pp. 127–143. Springer (Jul 2015), *http://link.springer.com/chapter/10.1007/978-3-319-21542-6_9*
5. Berners-Lee, T., Connolly, D., Kagal, L., Scharf, Y., Hendler, J.: N3Logic: A logical framework for the World Wide Web. Theory and Practice of Logic Programming 8(3), 249–269 (2008)
6. De Roo, J.: Euler yet another proof engine (1999–2016), *http://eulersharp.sourceforge.net/*
7. Verborgh, R., Arndt, D., Van Hoecke, S., De Roo, J., Mels, G., Steiner, T., Gabarró Vallés, J.: The pragmatic proof: Hypermedia API composition and execution. Theory and Practice of Logic Programming (2016), *http://arxiv.org/pdf/1512.07780v1.pdf*
8. Verborgh, R., De Roo, J.: Drawing conclusions from Linked Data on the Web. IEEE Software 32(5), 23–27 (May 2015), *http://online.qmags.com/ISW0515?cid=3244717&eid=19361&pg=25*
9. Verborgh, R., Steiner, T., Van Deursen, D., Coppens, S., Gabarró Vallés, J., Van de Walle, R.: Functional descriptions as the bridge between hypermedia APIs and the Semantic Web. In: Proceedings of the Third International Workshop on RESTful Design. pp. 33–40. ACM (Apr 2012)